演習２：集合の知性を設計する

(05)　05/20

Ａ｜Unity環境の整備・簡単なルールの設計

(06) 05/27　(07)　06/03

Ｂ｜ボイドルール１・２・３の実装

(08)　06/10

Ｃ｜課題１：集合知の解析

(09)　06/17　(10)　06/24

Ｄ１｜ＳＩＲ（感染モデル）

(11)　07/01　(12)　07/08　(13)　07/15

Ｄ２｜課題２：マイルール・感染ルール・視点操作

(14-15)　07/22

Ｄ３｜発表（One-Minute Movie）

２つの修正

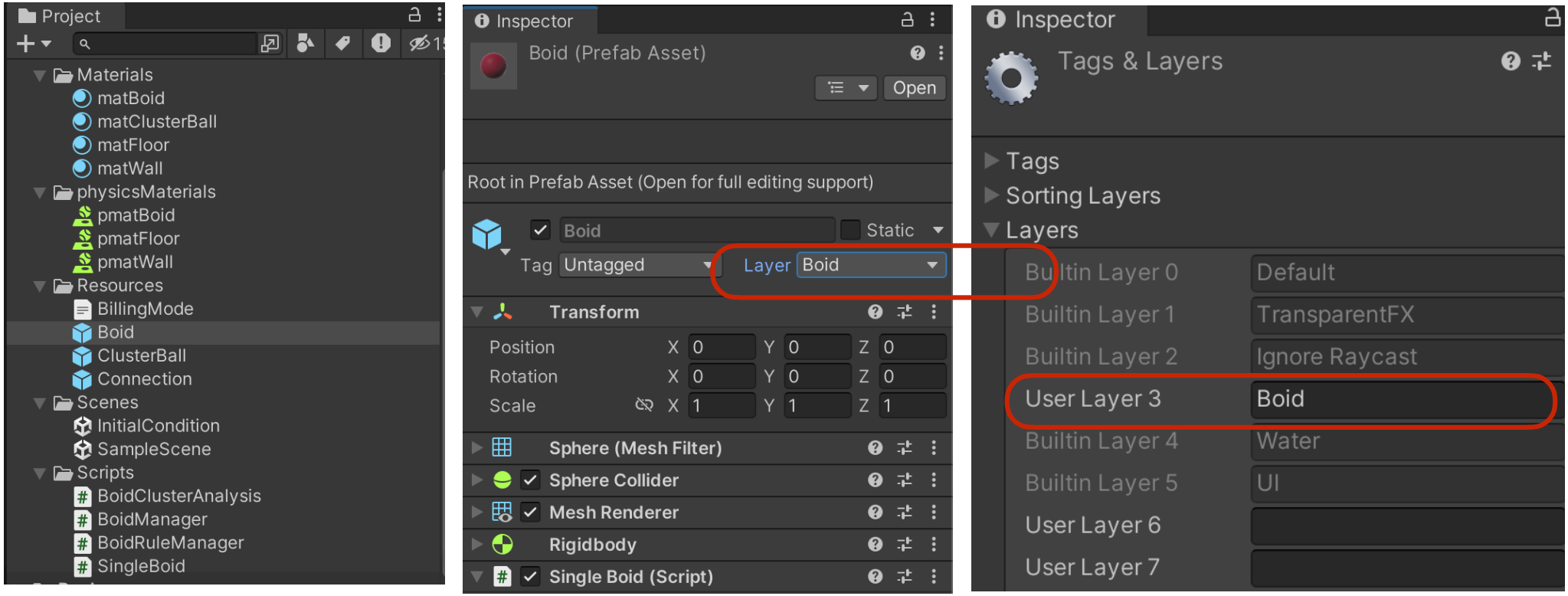
(1)　ボイド同士が透過するようにします。

(2)　壁への衝突直後の一定時間にルールの適用をキャンセルします。

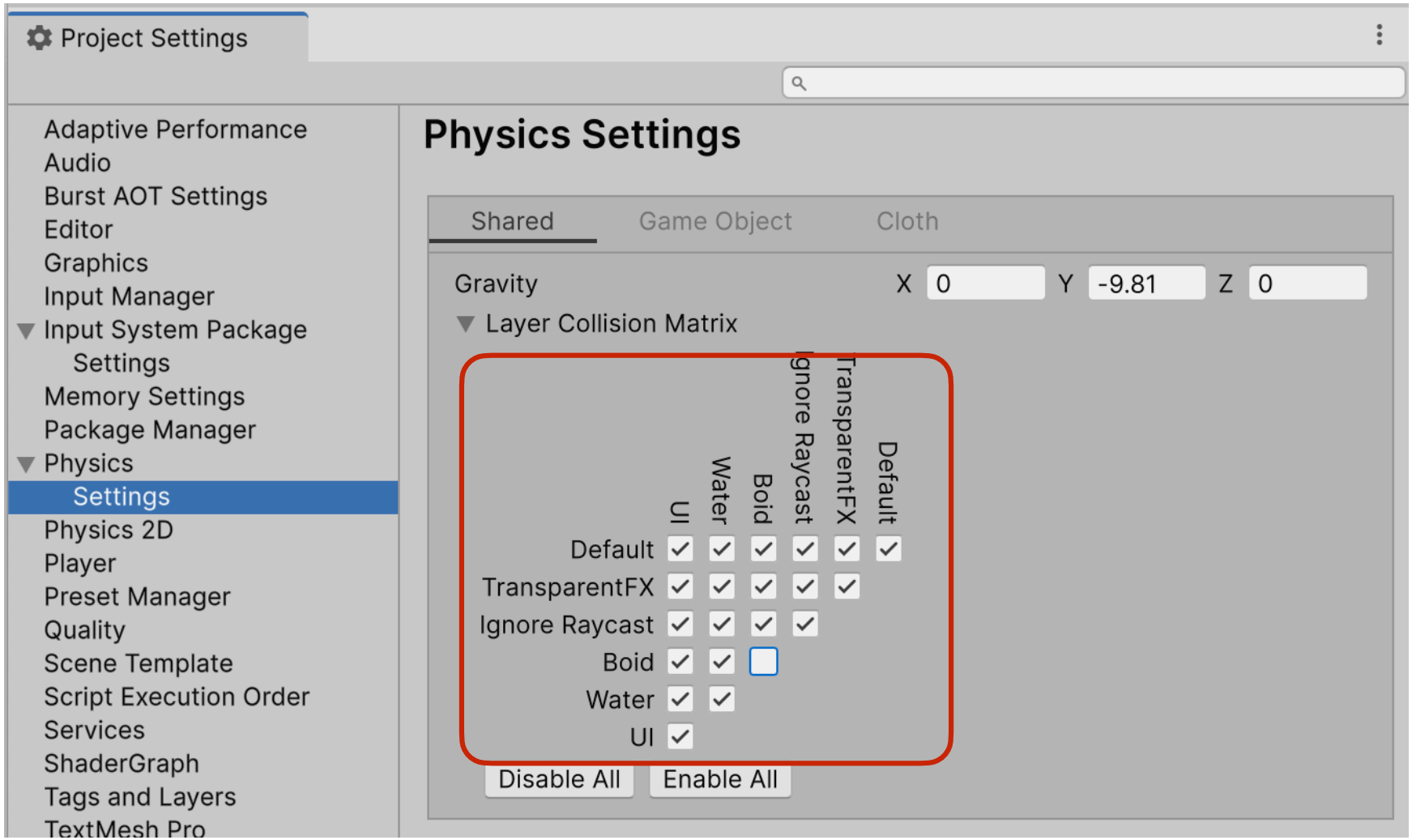
２つの修正

(1) ボイド同士が透過するようにします。

(2) 壁への衝突直後の一定時間にルールの適用をキャンセルします。



- 新たに「Boid」という名前のユーザ定義のレイヤを登録し、Builtin Layerの2と4の間に差し込みます。
- プレハブのボイドのレイヤを「Boid」に設定します。

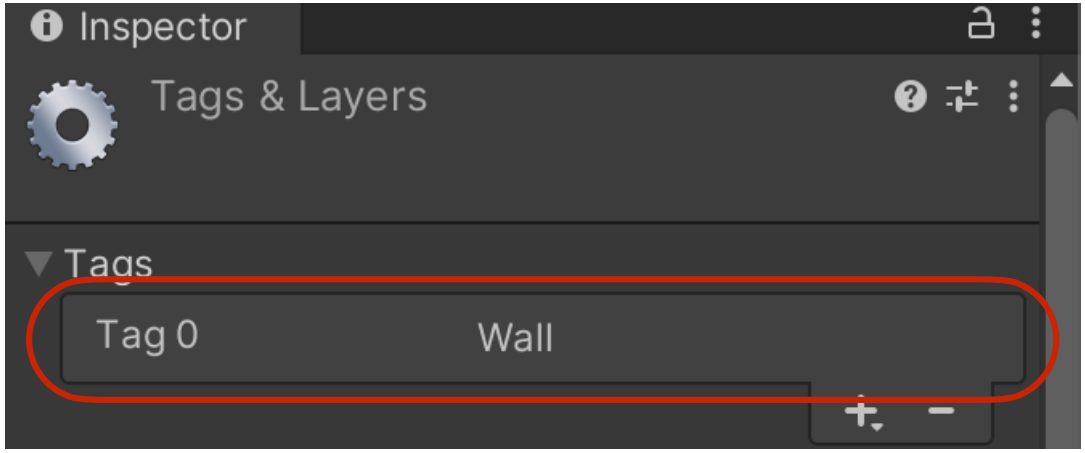


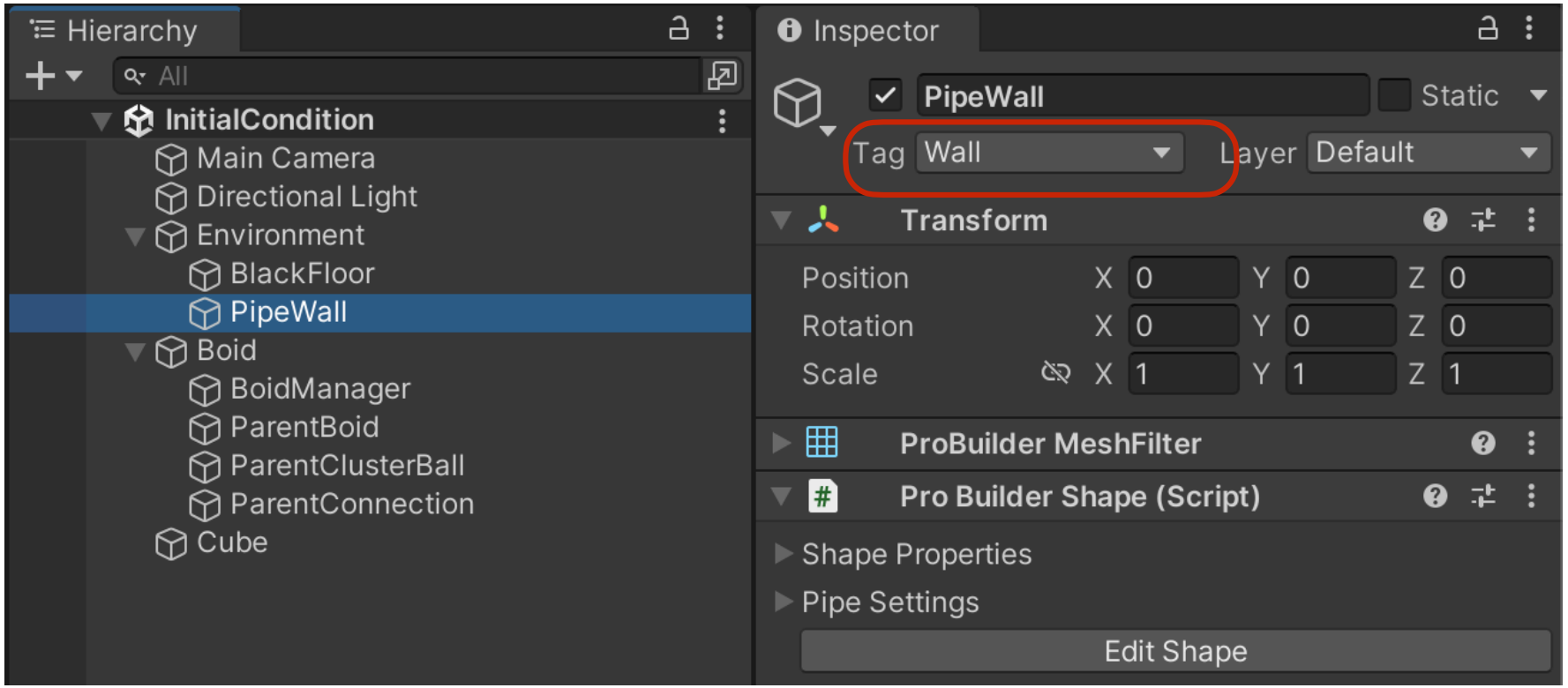
- メニューから「Edit→Project Settings→Physics→Settings」で、Sharedタブの Layer Collision Matrixを展開すると、レイヤ間のコリジョンの発生の有無を指定できます。ここでは、ボイド同士のコリジョンをオフにします。

２つの修正

(1) ボイド同士が透過するようにします。

(2) 壁への衝突直後の一定時間にルールの適用をキャンセルします。





- 新たに「Wall」という名前のタグを登録しPipeWallのタグとして割り当てます。

SingleBoid.cs

```
12      /* 壁への衝突 x 相互作用の無効 （2024.6.4）*/
13
14      //相互作用の無効（trueのときsetVelocityを機能させない）
        3 個の参照
15      public bool blindness = false;
16
17      //壁に衝突してからの時間（sec）
        3 個の参照
18      private float timeW = 0f;
19
20      //衝突後ボイドルールを無効にする時間（sec）
        1 個の参照
21      private float blindtime_after_wallhit = 0.8f;
22
```

宣言部

Update()

```
        0 個の参照
43      void Update ()
44      {
45          //位置と速度の更新
46          pos = this.transform.position;
47          vel = this.rb.velocity;
48
49          Rebound ();              //境界判定
50          LimitVelocity ();        //速度制限
51          setGravity();            //重力の設定
52
53          manageWall(); //壁との衝突管理
54
55      }
56
57      //壁に衝突後、blindnessをtrueとし、
58      //一定時間経過後、blinednessをfalseに戻す。
        1 個の参照
59      private void manageWall(){
60
61          timeW += Time.deltaTime;
62          if(timeW < blindtime_after_wallhit){
63              blindness = true;
64          }else{
65              blindness = false;
66          }
67
68      }
```

manageWall()

SingleBoid.cs

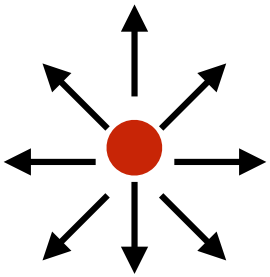
SetVelocity(v)

```
158     //速度の設定
        0 個の参照
159     public void SetVelocity(Vector3 v){
160
161         //blindnessがtrueでなければ速度を更新する。 (2024.6.24)
162         if(!this.blindness){
163             this.rb.velocity = v;
164             this.vel = this.rb.velocity;
165         }
166
167     }
```

```
181     //壁への衝突を検知すると、timeWを0に更新。
        0 個の参照
182     void OnCollisionEnter(Collision hit)
183     {
184         if(hit.gameObject.tag == "Wall")
185         {
186             timeW = 0;
187         }
188     }
```

⚠Onは大文字！！

OnCollisionEnter()



指定された点から
離れるようにするためのベクトル計算

位置による反発作用（斥力）

演習２-Ｂ

ボイドルールの設計（ルール２・ルール３）

衝突回避（ルール２）　整列行動（ルール３）

```
//ルールの係数
public float c1 = 0.1f;
public float c2 = 5.0f;
public float c3 = 0.01f;
```

```
//ルールの適用
public bool rule1 = false;
public bool rule2 = false;
public bool rule3 = false;
```

```
//ルール2の適用
if (rule2) {
    ApplyRule2 ();
}
//ルール3の適用
if (rule3) {
    ApplyRule3 ();
}
```

ApplyRules()

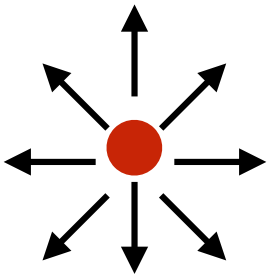
BoidRuleManager.cs

```
//ボイドルールマネージャ
BoidRuleManager rule;
```

```
//ボイドルールマネージャによるルールの適用
rule.SetBoid(this);
rule.ApplyRule();
```

Update()

BoidManager.cs

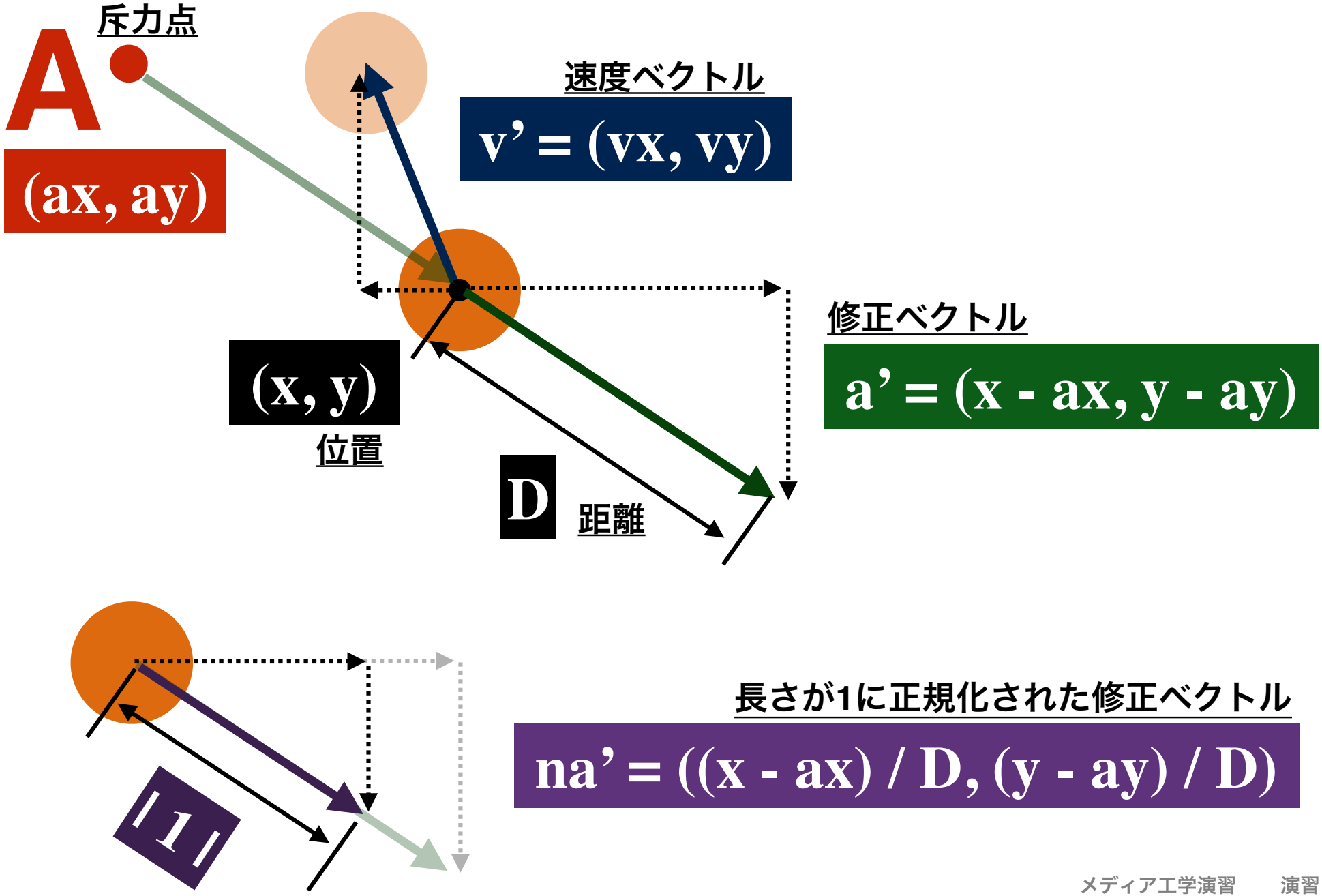


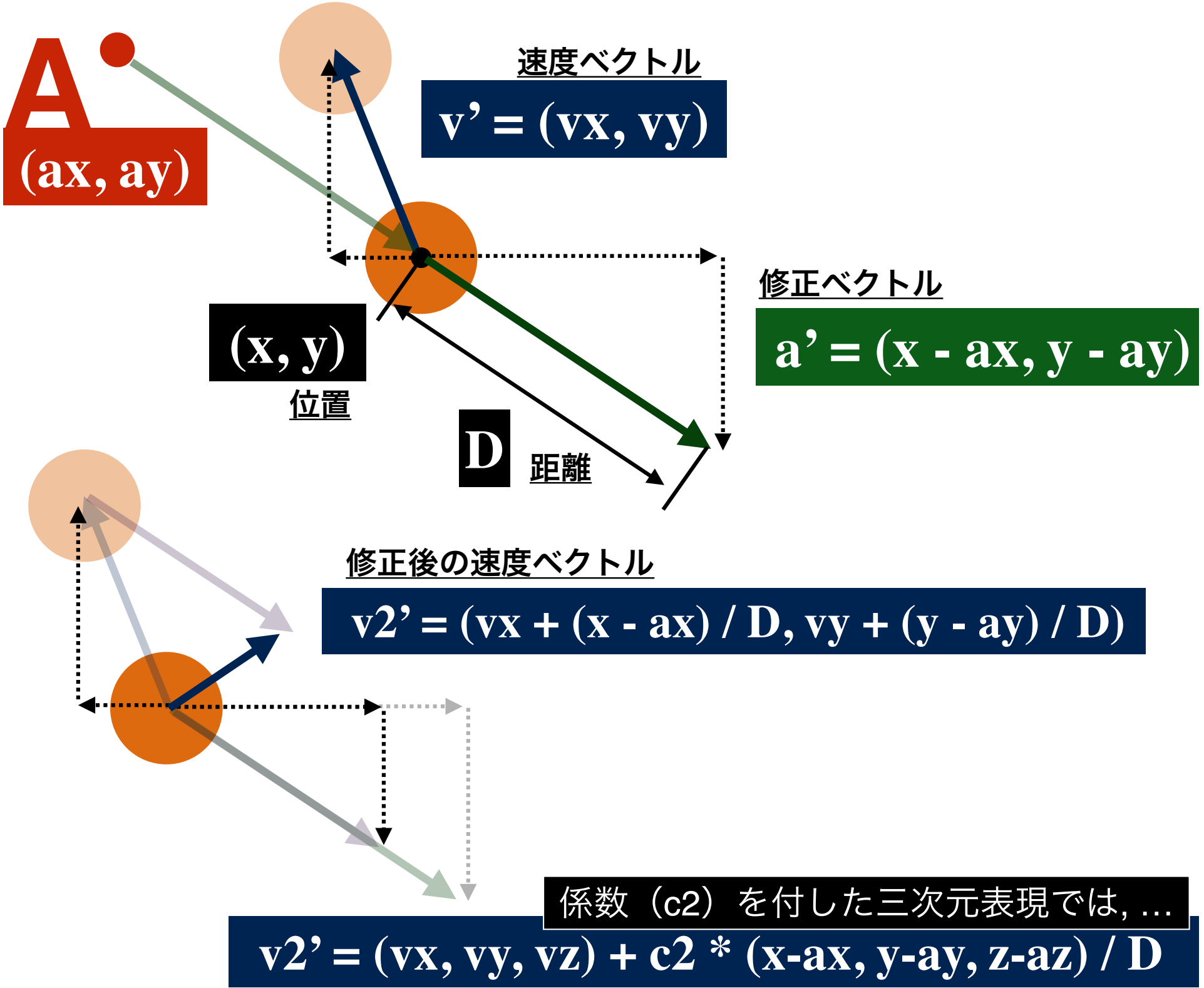
指定された点から
離れるようにするためのベクトル計算

位置による反発作用（斥力）

<速度 v'> を<点 A> から離れる方向に修正する

（今回は, 修正ベクトルの絶対値を1に正規化します）




A
(ax, ay)
速度ベクトル
v’ = (vx, vy)
(x, y)
位置
修正ベクトル
a’ = (x - ax, y - ay)
D
距離
修正後の速度ベクトル
v2’ = (vx + (x - ax) / D, vy + (y - ay) / D)
係数（c2）を付した三次元表現では, …
v2’ = (vx, vy, vz) + c2 * (x-ax, y-ay, z-az) / D

ルール２（反発）

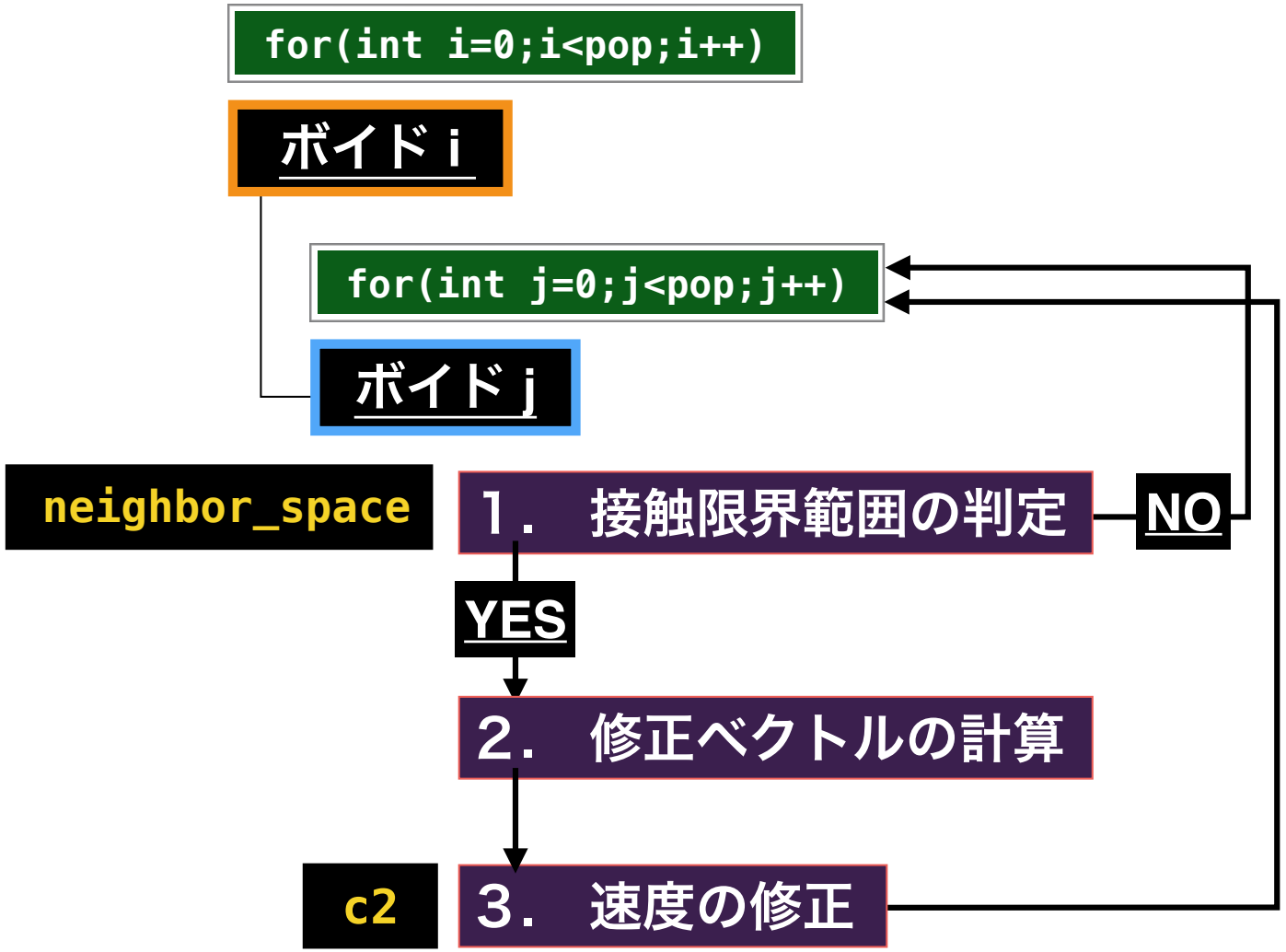
BoidRuleManager

- <int> pop
  - ボイドの個体数
- <float> c2
  - 斥力の強さ

SingleBoid

- <Vector3> pos
  - 各ボイドの三次元位置
- <Vector3> vel
  - 各ボイドの速度
- <float> neighbor_space
  - 各ボイドの接触限界距離
- void SetVelocity(Vector3 v)
  - 速度更新メソッド

ルール２（分離）の実装

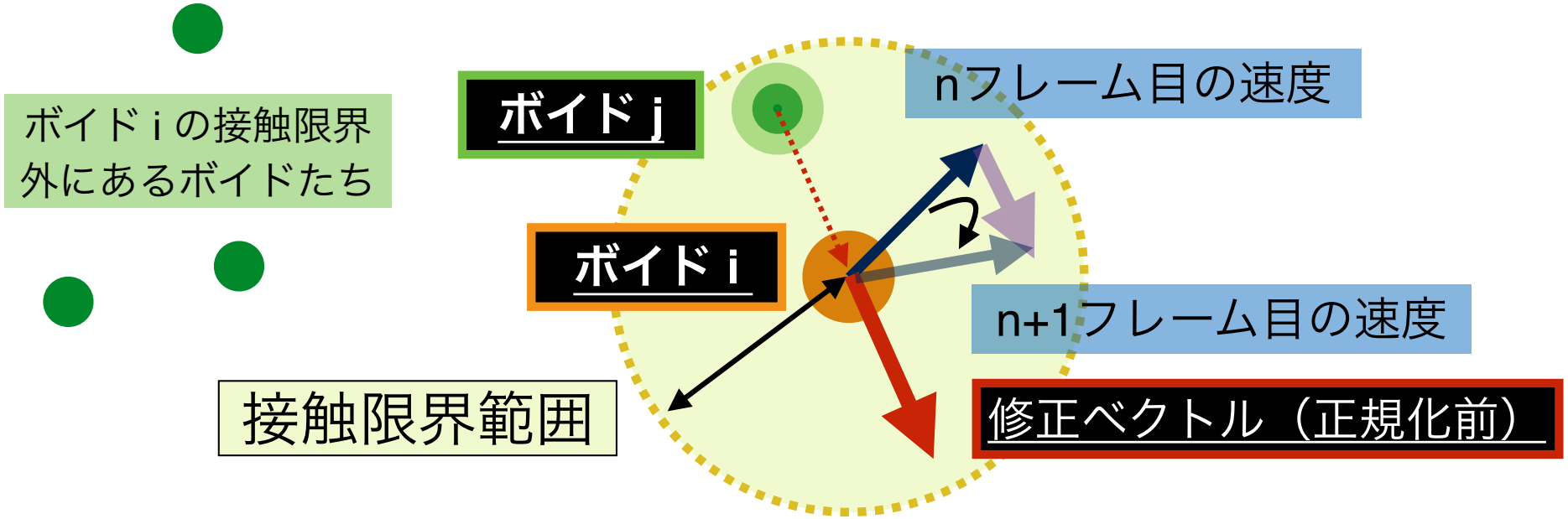


上記のフローのように，（平均を求めるような手法ではなく）
逐次的に速度を修正する手法が最も効率的な分離を可能とするようです．

ルール２（反発）

ルール２（分離）の実装

１．ボイド i に関する接触限界範囲の判定



ボイド j が ボイド i の 接触限界範囲 にあるか？

→

YES

２．修正ベクトルの計算

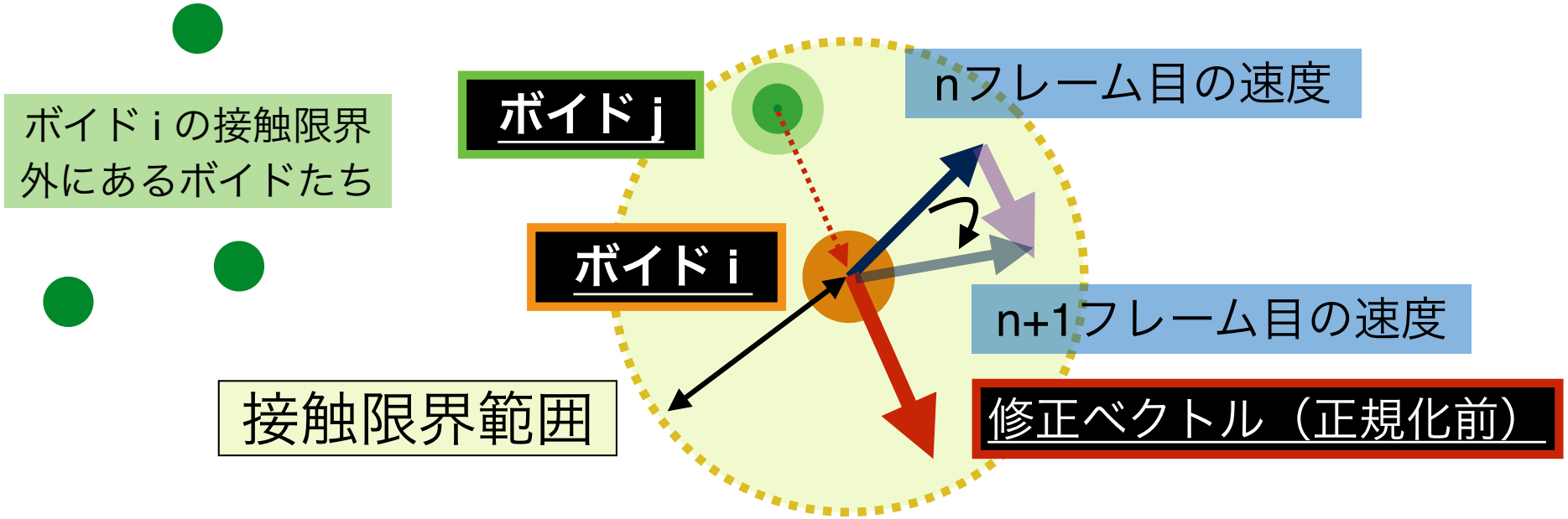
NO

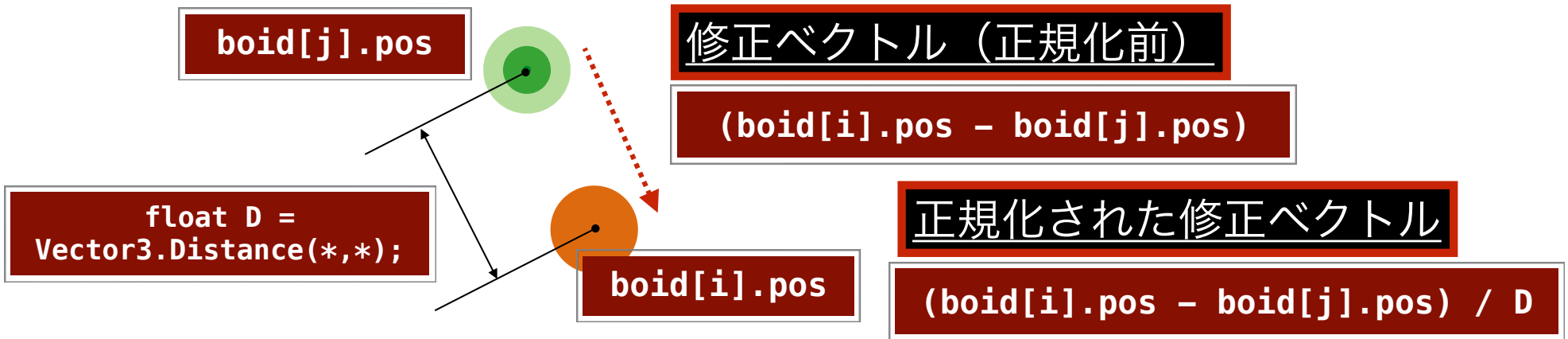
ボイド j+1 の １．接触限界範囲の判定 へ

ルール２（反発）

ルール２（分離）の実装

２．修正ベクトルの計算





反発ルール（ルール２）関数内部の記述例

全てのボイド（i=0…pop−1）について，
1) 接触限界範囲内にあるボイドを見つけ，
2) 引数である係数c2を用いて反発する方向に速度を修正します．

```
private void ApplyRule2()
{
    for(int i=0;i<pop;i++){
        Vector3 ipos = boid[i].pos;
        float ineighbor_space = boid[i].neighbor_space;

        for(int j=0;j<pop;j++){
            Vector3 jpos = boid[j].pos;
            float dis = Vector3.Distance(ipos,jpos);

            if(
                }
        }
    }
}
```

i 番目のボイドの位置をiposとする．

i 番目のボイドの接触限界範囲を ineighbor_spaceとする．

j番目のボイドの位置をjposとする．

ボイドiとボイドjの距離を disとする．

ineighbor_spaceを使って条件を書きます．

１．　接触限界範囲の判定

２．　修正ベクトルの計算

係数 c2 、ipos・jpos・disを使って、ボイドiの速度を更新．

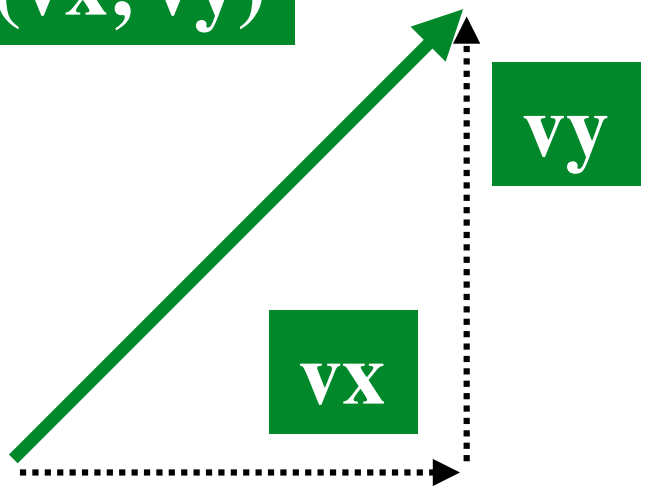
BoidRuleManager.cs

速度を, 別の速度（マスターベクトル）に徐々に向けるためのベクトル計算

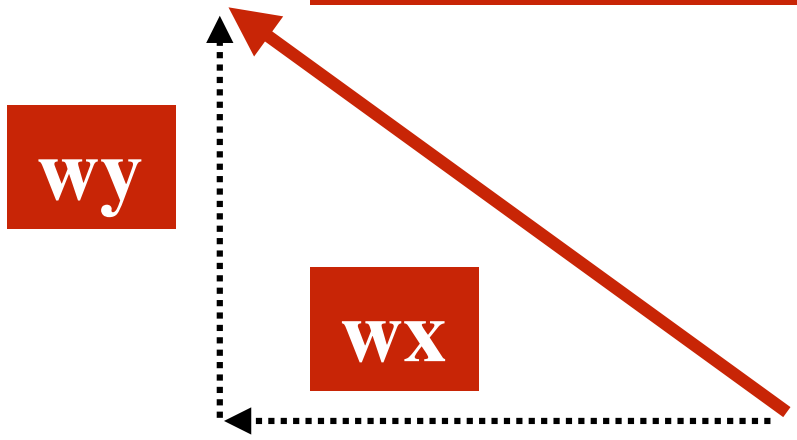
速度による引き込み
（内分ベクトルによる方法）

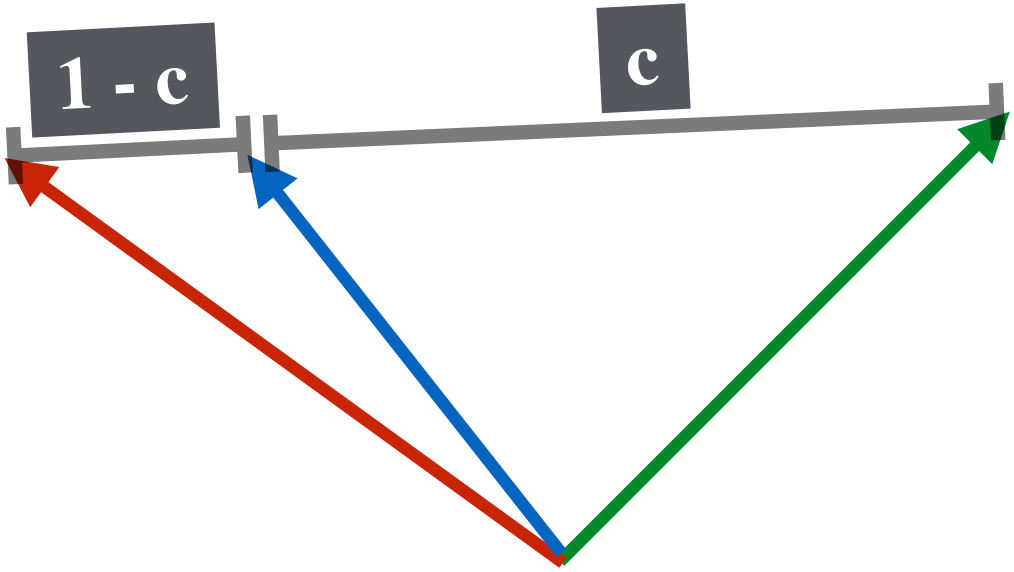
内分ベクトルの計算

v’ = (vx, vy)



w’ = (wx, wy)





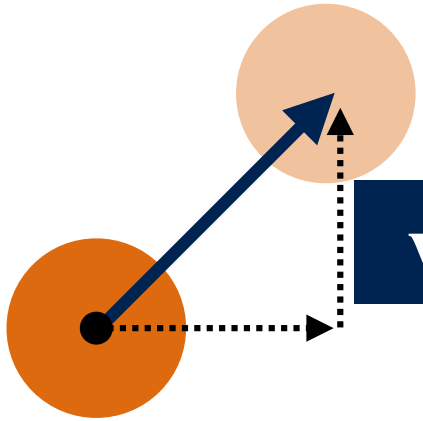
v’とw’を c:1-c に内分するベクトル

(c*wx + (1 - c)*vx, c*wy + (1 - c)*vy)

<速度 v'> を<速度w'> の方向に修正する（内分ベクトルによる方法）

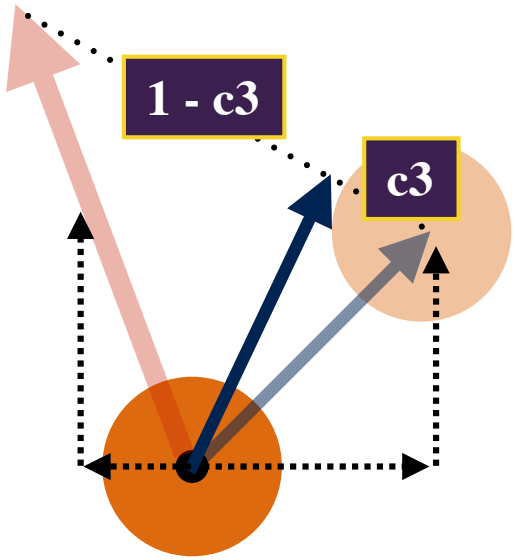
マスターベクトル

w' = (wx, wy)



もともとの
速度ベクトル（v0）

v0' = (vx, vy)

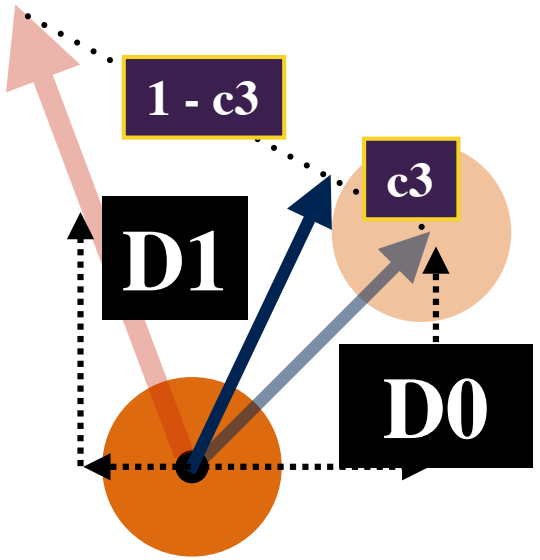


更新された速度ベクトル（v1）

v1' = (c3*wx+(1-c3)*vx,
c3*wy+(1-c3)*vy)

c3 が **1** に近い程, すぐマスターベクトルに引きこまれる.

c3 が **1** に近い程, すぐマスターベクトルに引きこまれる.



更新された速度ベクトル（v1）

$$v1' = (c3*wx+(1-c3)*vx,\ c3*wy+(1-c3)*vy)$$

速さを調整した速度ベクトル（v2）

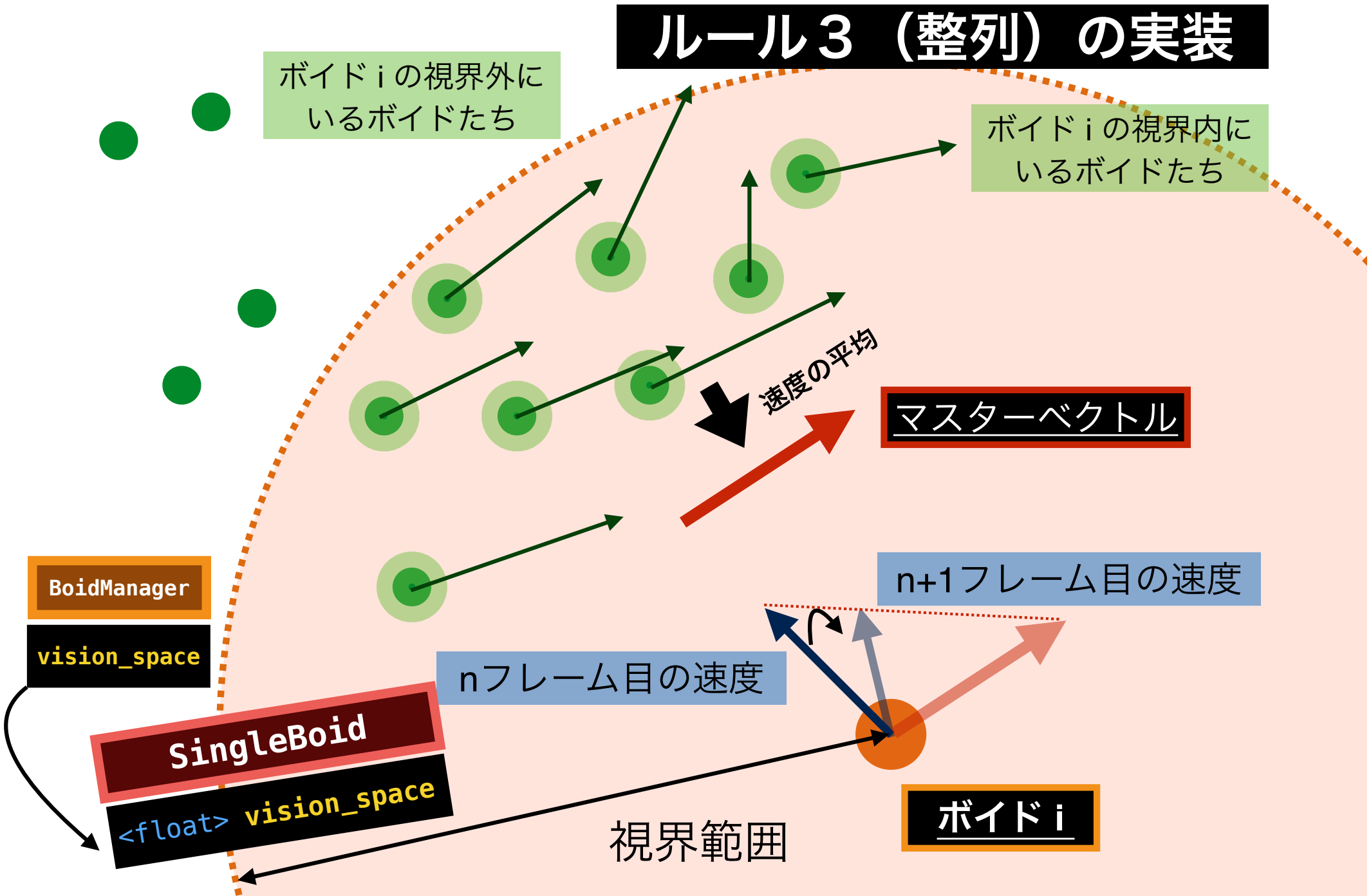
$$v2' = (D0 / D1) * v1'$$

D0 ：v0’の距離

D1 ：v1’の距離

速度ベクトルの距離（速さ）を、D0（もともとの速さ）に調整する。

ルール３（整列）


ルール３（整列）の実装
ボイドｉの視界外にいるボイドたち
ボイドｉの視界内にいるボイドたち
速度の平均
マスターベクトル
n+1フレーム目の速度
nフレーム目の速度
BoidManager
vision_space
SingleBoid
<float> vision_space
視界範囲
ボイドｉ

ルール３（整列）の実装

マスターベクトルの計算

SingleBoid

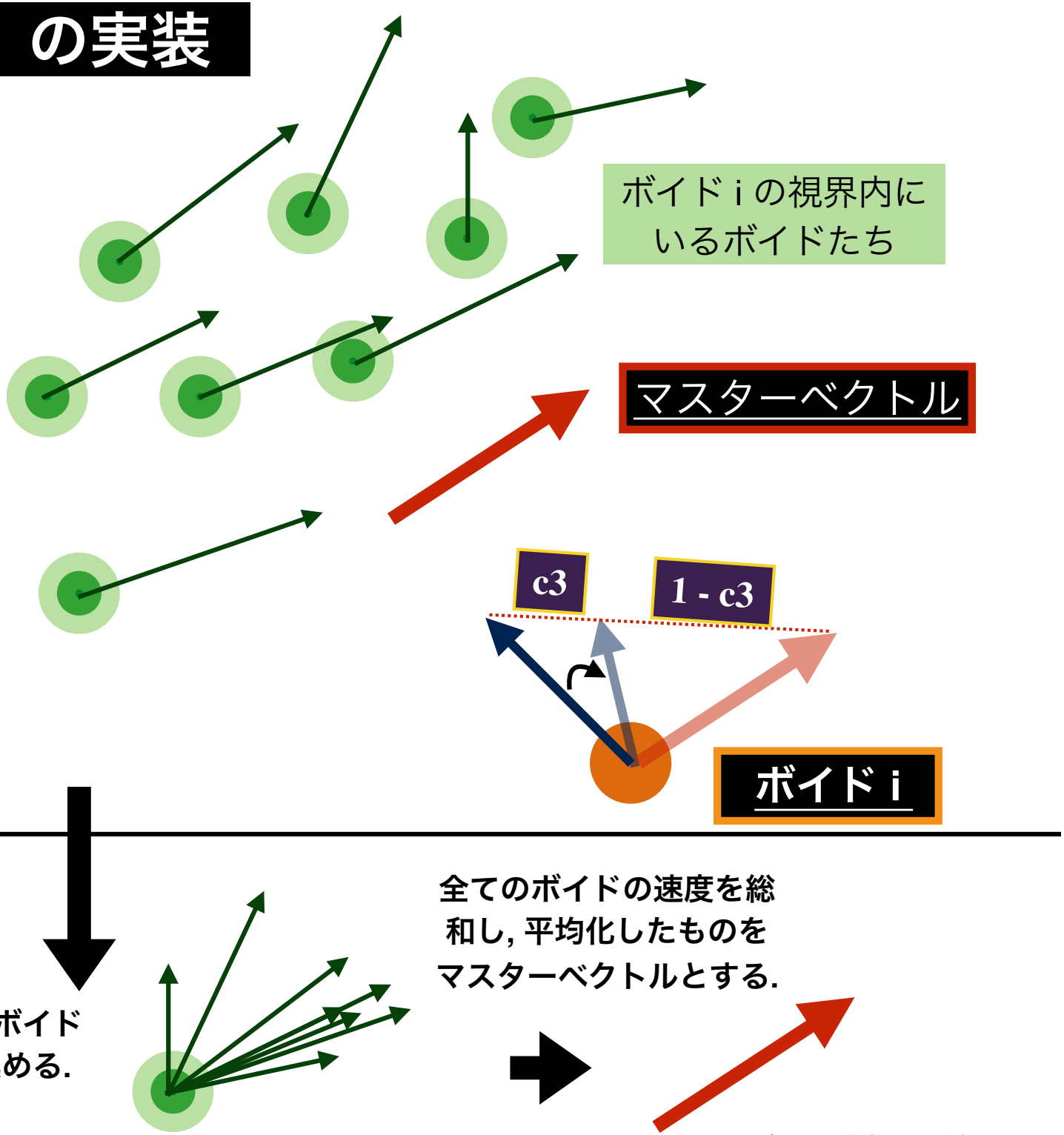
`<float> vision_space`

各ボイドの視界距離

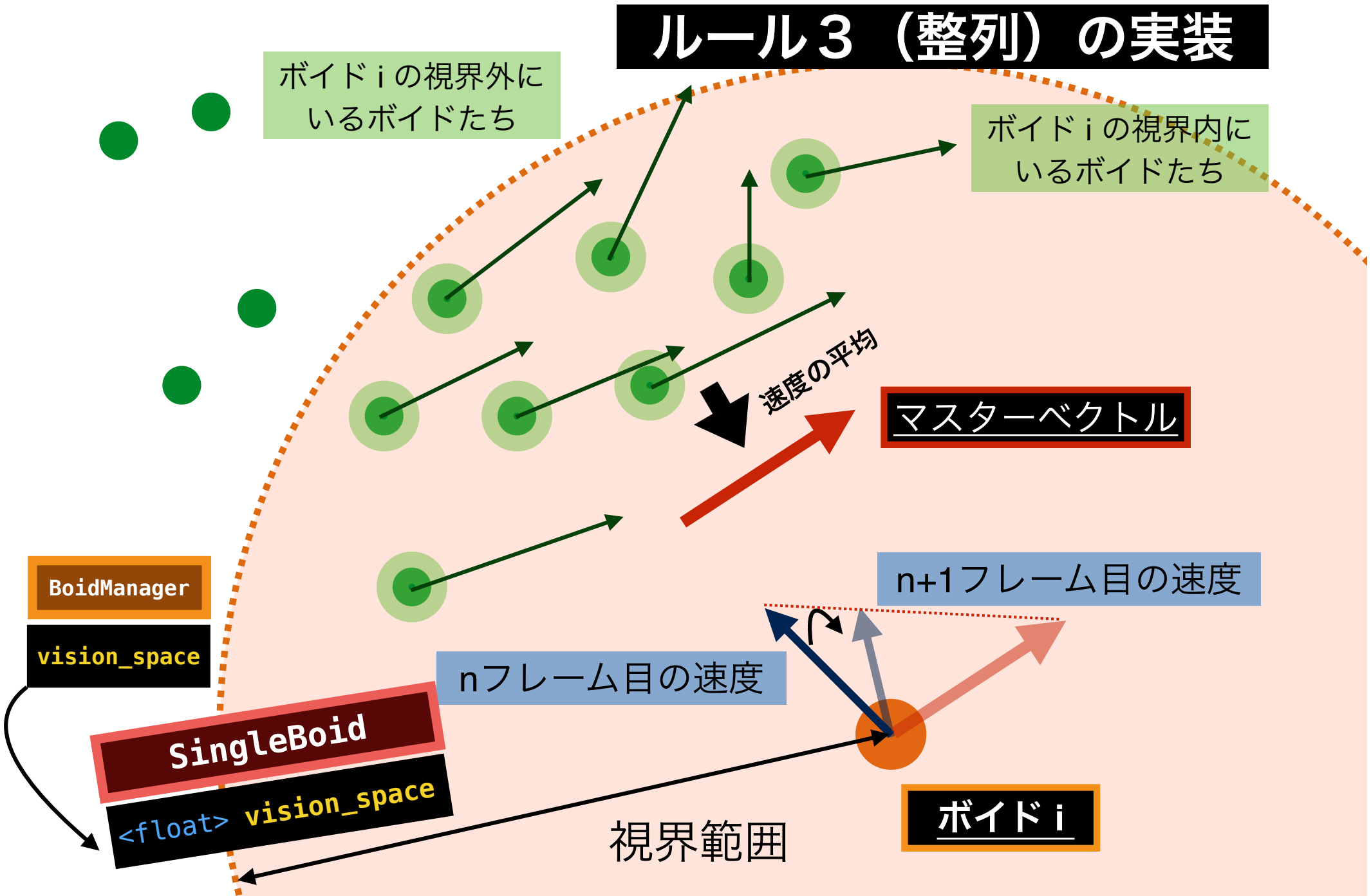
BoidRuleManager

`<float> c3`

整列の引き込みの強さ



ルール３（整列）


ルール３（整列）の実装
ボイドｉの視界外にいるボイドたち
ボイドｉの視界内にいるボイドたち
速度の平均
マスターベクトル
n+1フレーム目の速度
nフレーム目の速度
BoidManager
vision_space
SingleBoid
<float> vision_space
視界範囲
ボイドｉ

整列ルール（ルール３）関数内部の記述例

```
private void ApplyRule3()
{
    for(int i=0;i<pop;i++){
        Vector3 ipos = boid[i].pos;
        Vector3 ivel = boid[i].vel;
        float ivision_space = boid[i].vision_space;

        float count = 0;
        Vector3 velSum = Vector3.zero;

        for(int j=0;j<pop;j++){
            Vector3 jpos = boid[j].pos;
            Vector3 jvel = boid[j].vel;

            float d = Vector3.Distance(ipos,jpos);

            if(j!=i &&                    ){

                }
            }

        if(count>0){

            boid[i].SetVelocity(
        }
    }
}
```

i 番目のボイドの位置・速度・視界距離をipos, ivel, ivision_spaceとする.

count: 視界内にいるボイドの数

velSum: 視界内ボイドの速度総和

j番目のボイドの位置をjposとする.

dに関する条件が入ります。

count, velSumに対する処理を書きます.

係数 c3 を使って、i 番目のボイドの速度を更新します。

BoidRuleManager.cs